Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 65 (2007) 249–274
Science of Computer Programming
www.elsevier.com/locate/scico

Odyssey-SCM: An integrated software configuration management infrastructure for UML models☆

Leonardo Murta*, Hamilton Oliveira, Cristine Dantas, Luiz Gustavo Lopes, Cláudia Werner

COPPE/UFRJ–Systems Engineering and Computer Science Program, Federal University of Rio de Janeiro–P.O. Box 68511, 21945-970 Rio de Janeiro, Brazil

Received 1 December 2005; received in revised form 30 March 2006; accepted 15 May 2006
Available online 28 November 2006

Abstract

Model-driven development is becoming a reality. Different CASE tool vendors support this paradigm, allowing developers to define high-level models and helping to transform them into refined models or source code. However, current software configuration management tools use a file-based data model that is barely sufficient to manipulate source code. This file-based data model is not adequate to provide versioning capabilities for software modeling environments, which are strongly focused on analysis and architectural design artifacts. The existence of a versioned repository of high-level artifacts integrated with a customized change control process could help in the development and maintenance of such model-based systems. In this work, we introduce Odyssey-SCM, an integrated software configuration management infrastructure for UML models. This infrastructure is composed of a flexible version control system for fine-grained UML model elements, named Odyssey-VCS, and two complementary components: a customizable change control system tightly integrated with the version control system, and a traceability link detection tool that uses data mining to discover change traces among versioned UML model elements and provides the rationale of change traces, automatically collected from the integrated software configuration management infrastructure.

Keywords: Software configuration management; Model-driven development; Version control system; Change control system; Data mining

1. Introduction

Computer Aided Software Engineering (CASE) tools can be classified into two main groups [48]: lower CASE tools and upper CASE tools. Lower CASE tools are mostly concerned about implementation and testing issues; whereas upper CASE tools deal with higher abstraction levels, entailing requirements, analysis, and design artifacts. Despite the necessity of Software Configuration Management (SCM) support for every kind of CASE tool [42], this

☆ This paper is an extended version of the paper "Odyssey-VCS: A Flexible Version Control System for UML Model Elements", which was published at the 12th international workshop on software configuration management.

* Corresponding author.
E-mail addresses: murta@cos.ufrj.br (L. Murta), hamilton@cos.ufrj.br (H. Oliveira), cristine@cos.ufrj.br (C. Dantas), luizgus@cos.ufrj.br (L.G. Lopes), werner@cos.ufrj.br (C. Werner).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.05.011

support has been typically focused on lower CASE tools, providing a huge infrastructure over the last decades to leverage evolution of source code artifacts.

However, due to the increasing software development complexity, SCM support is also needed by upper CASE tools. Model-driven development is emerging as a promising technique for complexity control. Model-driven approaches focus on the definition of high-level models and apply subsequent transformations to obtain implementation artifacts [3]. Nevertheless, the current SCM infrastructure does not properly support the evolution of model-based artifacts.

A first thought would be to adapt the existing SCM techniques, formerly applied to source code, to this new context. However, current SCM infrastructures are not suited to the fine-grained artifacts used by upper CASE tools. These SCM infrastructures were intentionally designed to be generic [16], avoiding language-specific support. For example, most current SCM systems are based on file system structures, while upper CASE tools are based on higher-level structures. The mapping of these complex structures used by upper CASE tools to file structures is dangerous due to concept mismatch. Nevertheless, the SCM community has already detected that many unaddressed research issues rely on breaking the assumption of generic and language-independent SCM [16].

Moreover, most SCM standards [25,26] recommend the selective identification of Configuration Items (CI) that depend on individual characteristics of the software development projects. Nearly all state-of-the-practice SCM systems have a fixed identification of CI: the file. Due to this fact, every artifact that needs versioning information should be stored into an individual file. However, in some circumstances it is neither desirable nor possible to map every high-level analysis and design artifact into an individual file.

Finally, even when a SCM repository is in place, software engineers must ensure that all artifacts are up to date and consistent throughout the different abstraction levels to avoid misunderstandings. For this reason, it is important to identify, for example, UML model elements that shall be updated when other UML model elements are changed. This can help in avoiding incomplete changes, detecting hidden dependencies, supporting impact analysis, and suggesting likely further changes [53].

Aiming to diminish the effects of these problems, we propose a novel approach to support UML-based upper CASE tools in evolving UML models. This approach, called Odyssey-SCM, consists of a Version Control System (VCS) for fine-grained UML model elements, named Odyssey-VCS [36], which can be tailored to the specific needs of a software development project, as recommended by SCM standards. Moreover, Odyssey-SCM also encloses two complementary components: a customizable Change Control System (CCS), named Odyssey-CCS [31], tightly integrated with Odyssey-VCS, and a traceability link detection tool [10] that uses data mining to discover change traces among versioned UML model elements and provides the rationale of change traces, automatically collected from the integrated software configuration management infrastructure. The main goal of our approach is to aid architects in the concurrent development and evolution of model-based software systems using heterogeneous UML-based upper CASE tools.

Odyssey-VCS maintains a per-project behavior descriptor that informs how each UML model element type should be dealt with. This behavior descriptor determines when evolution information is needed for a UML model element, considering this element as a CI. The evolution information comprises a unique version identification and auxiliary contextual information, such as who changed the element, when it was changed, and why it was changed. Moreover, this behavior descriptor also indicates which elements are considered atomic for conflict detection purposes. Odyssey-VCS raises a conflict when two or more developers try to concomitantly check-in changes over an element that is considered atomic.

Odyssey-CCS allows configuration managers to model the change control process (CCP) and to define which pieces of information need to be collected by each activity of the modeled CCP. This information collection is done via templates and fields. Each activity is associated to a set of templates, and each template is composed of a set of fields. For a given template and field, Odyssey-CCS is able to inform the value that was filled in during the execution of a specific CCP activity.

Our traceability link detection tool searches for patterns, called change traces, in the Odyssey-SCM repository. These patterns indicate that some UML model elements are usually changed when another UML model element is changed. Due to the integrated SCM infrastructure, it is possible to contextualize the detected change traces with information filled in the Odyssey-CCS templates.

During the design of our approach, we provided our own solutions to overcome some challenges described in the SCM literature [14], such as: (1) a data model that deals with complex CIs; (2) homogeneous versioning for different

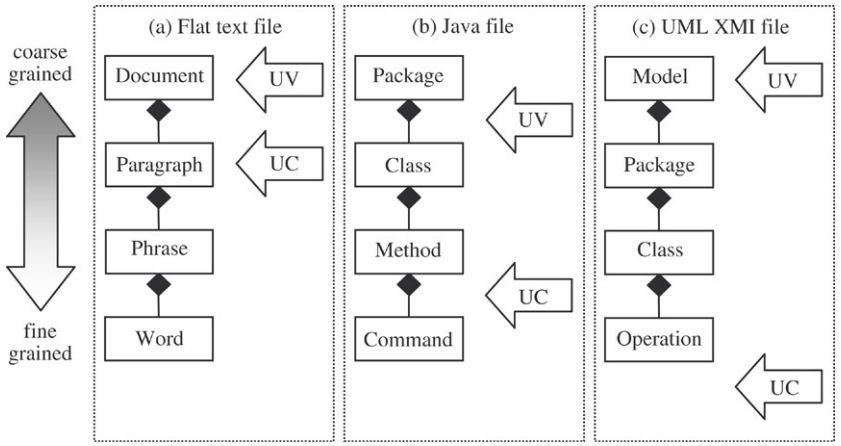

Fig. 1. Current UC applied to flat text files, Java files and UML XMI files.

types of CIs; (3) distributed and heterogeneous workspaces; and (4) concurrent engineering with high-level models. Moreover, a guiding philosophy of our work is to adopt standardized solutions and successful technologies used in other SCM systems.

The rest of this paper is organized as follows. Section 2 details the problem to ground the ensuing discussion. Section 3 presents an overview of Odyssey-VCS, which is followed by a discussion of its internal mechanisms in Section 4. Section 5 introduces some complementary work: components built over Odyssey-VCS for change control and traceability link detection. Section 6 shows a straightforward example that demonstrates how the challenges formerly discussed are addressed. Section 7 presents an initial evaluation of Odyssey-VCS. Section 8 discusses related work, and we conclude the paper in Section 9 with an outlook on our future work.

2. Problem statement

VCSs that use a data model based on file system structures usually consider files and directories as their CIs. This approach leads to three types of CIs: composite, textual and binary. The first type, composite CI, is represented by directories and can aggregate textual, binary or other composite CIs. The second type of CI, represented by text files, is the most important type because it can be internally manipulated by the VCS in order to execute basic version control operations, such as diff, patch and merge. The third type, binary files, is only controlled, but not internally manipulated by the VCS since their internal structures are usually opaque to this tool.

For this reason, text files are seen by these VCSs as white box artifacts, and binary files are seen as black box artifacts. In fact, the use of automatic merge facilities, even for text files, is known to be an error prone activity [30]. They usually provide generic merging algorithms that do not take into account the specific syntactic structure of the text file. Despite this limitation, almost all VCSs use text file mergers without differentiating text file contents. For example, the same merge algorithm used on a flat text file is also used on a Prolog file, a Java file, a LaTeX file or even on an XML file. It is important to mention that the usual text file mergers proved to make a good job, despite their weakness. By being generic, they helped to propagate source code SCM independently of the adopted programming language [16].

Moreover, a common algorithm used by many tools to discover the type of a file is based on control characters. This algorithm searches for a zero ASCII byte inside the file. If this byte is found, the file is marked as binary; otherwise, the file is marked as text. This algorithm is used, for example, by MS Visual SourceSafe [44]. Beyond other problems related to the non-existence of zero ASCII byte control code in some binary files, all different kinds of text files will be marked as a flat text file. Nevertheless, few VCSs such as Rational ClearCase provide special support for different file types (eg. mdl, doc, xml, etc.) and allow the usage of external mergers [50].

When a file is marked as a flat text file, the VCS considers a line as the unit of comparison[1] (UC). As shown in Fig. 1(a), the UC used in a flat text file is mapped into a paragraph. This is a well fitted mapping because a paragraph

[1] We define unit of comparison as an atomic element used for conflict computation. Conflicts occur when two or more developers concurrently work on any part of the same unit of comparison.

has enough cohesion and relative low coupling with other paragraphs, and a defined structure composed of a topic sentence and some supporting sentences. However, this is not the case in other situations as follows:

- A Prolog file usually has complex facts and rules written using more than one line. In this case, the UC should be Prolog predicates.
- A Java file, as any other object-oriented language, has complex structures such as packages, classes, methods, and attributes, with methods and attributes as possible candidates to be the UC.
- A LaTeX file does not use carriage return and linefeed as delimiters to a paragraph structure. A blank line is needed to identify a paragraph. As a consequence, the UC should be the whole structure between blank lines.
- Finally, an XML file is a document composed of elements that may have other elements and attributes by themselves. In this context, a reasonable UC would be elements or attributes. An element or an attribute may be composed of many lines and changes in any of these different lines should be considered as changes in the same UC.

The use of line as the UC is especially applicable in situations when a single line has high cohesion and low coupling with other lines. Lines should not be used as the UC in files that employ data models with different abstraction structures. In the case of Java files, if a line metaphor is used, the UC is mapped to a non-existing Java structure (Fig. 1(b)). A Java method may be implemented by more than one line; whereas a line may comprise more than one Java command ended by a semicolon. In addition, a command is usually excessively coupled to other commands to be considered the UC. Hence, the indicated structure should be Java attributes and methods.

As discussed before, some VCSs, such as Rational ClearCase, provide support for different abstraction structures through a pluggable merge facility. This strategy, however, is not enough because the UC concept only helps to set a boundary between file parts. In these systems, the whole file is considered to be the unit of versioning[2] (UV). For example, in a flat text file, the UC is a paragraph and the UV is the whole document, as shown in Fig. 1(a).

This problem is more significant when the file type does not fit the abstraction structure of the file system data model. For example, Java may have one or more classes per file and these classes must be part of the same package, consequently, the UV maps to a non-existing Java element, as shown in Fig. 1(b). In this scenario, a package may have more than one UV distributed through different files, and a class may share the same UV with other classes in the same file.

Java files are not the worst case. A convention may be established to recommend the construction of only one class per file. In this way, the UV would be mapped to the class abstraction. Therefore, each class in the system would have their evolution controlled individually. On the other hand, object-oriented data model can also be made persistent through files. One of the most common approaches to map an object-oriented data model to a file is to use markup languages. In this case, the whole object network is mapped into a single file: an XML file. For instance, when Rational ClearCase marks a file as XML, the UC is changed and the merge and diff tools act in a special way to provide more control over parallel development. However, the UV remains being the whole XML file.

In our specific case, UML-based upper CASE tools use an object-oriented data model named Meta Object Facility (MOF) [37] and persist their models using XML Metadata Interchange (XMI) format [41]. In this scenario, the whole object network, composed of thousands of analysis and design artifacts, is persisted into a single XMI file. Fig. 1(c) shows a fragment of a UML class diagram mapped to an XMI file. The UC is smaller than an operation even if Rational ClearCase XML merge is used. This occurs because a UML operation is described via many XML elements, which represent visibility, return type, arguments, etc., and the parallel development over different XML sub-elements of the same UML operation would not raise conflicts.

Another bad aspect related to the UC in this scenario regards the proximity principle. UML uses an N-dimensional structure composed by different diagrams to model software, and this N-dimensional structure is mapped to a single XMI file, which is a one-dimensional structure. This problem leverages the difficulty of implementing generic conflict detection algorithms. For example, two classes connected via inheritance association are considered "near" in a UML model. However, if more than one developer changes these classes concurrently, the VCS would probably not be able to detect a conflict because the classes are apart from each other in the XMI file.

[2] We define unit of versioning as an atomic element associated to versioning information. A new version of the element is created when any of its parts is modified.

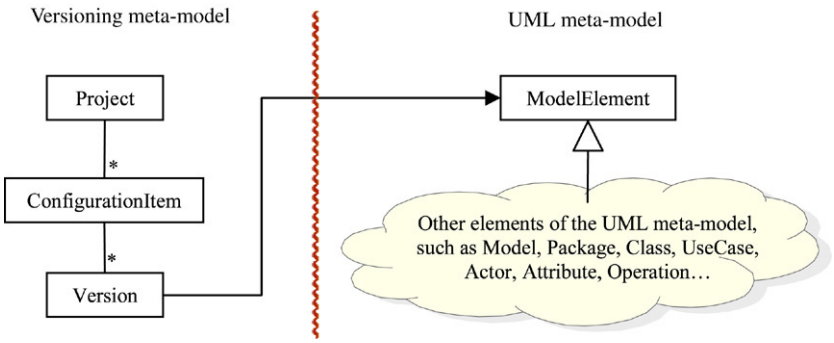

Fig. 2. Relationship between the versioning model and the data model.

The problems are even worse when analyzing the UV. The UV in this case is the whole UML model and it is not possible to distinguish versions of its parts, such as classes or use cases. The upper CASE tools will only be able to select versions of the whole model to work on. Therefore, the developers could not ask the VCS as to who changed a specific use case two days ago or what the existing versions of a given class are.

3. Odyssey-VCS overview

Aiming to diminish the effects of the problems presented in Section 2, we introduce Odyssey-VCS, a flexible VCS for UML model elements. In this section, we discuss the high-level features of Odyssey-VCS and show how they help to overcome the challenges presented in Section 1.

3.1. Complex data model

Every upper CASE tool splits modeling elements into two categories: semantic and syntactic elements. Semantic elements symbolize conceptual elements and contain all information related to these elements, while syntactic elements are representations of semantic elements inside a diagram and their data are diagram dependent, like color, position, and size. In the context of the proposed approach, CIs are semantic elements of UML-based upper CASE tools. To be more precise, any subtype of *ModelElement* in the UML meta-model [40] is a candidate to be a CI in our approach. For this reason, Odyssey-VCS is able to version even the relationships among UML model elements, since relationships are also model elements. Examples of these model elements are: use cases, actors, classes, class associations, operations, attributes, components, and so on.

Due to the complexity of this data model, it is not desirable to have a single versioning behavior for every CI type. Moreover, most SCM standards recommend the definition of a per-project SCM plan [25] and an important section of the SCM plan is the CI identification. The CI identification section of the SCM plan describes all CIs that should be placed under SCM. However, the current VCSs do not work with fine-grained CIs. As a consequence of this problem, all artifacts are put under version control, resulting in an extra overhead to the overall process, since some artifacts are not supposed to be controlled.

Our approach allows a fine-grained definition of CIs. For example, a class may be defined as an atomic CI for a given project; whereas operations and attributes may be controlled in another project. This flexibility provided by Odyssey-VCS allows more precise definition of CIs, adhering to the recommendations of existing SCM standards.

Besides the use of UML meta-model as a data model, Odyssey-VCS has its own versioning meta-model, which is also a MOF meta-model. In the Odyssey-VCS versioning meta-model, each instance of the *Project* type is composed of many instances of *ConfigurationItem* type, and each instance of *ConfigurationItem* type is associated to many instances of *Version* type. Correspondingly, each instance of *Version* type is associated to an instance of *ModelElement* type, which is a generic supertype of any UML model element. Due to that, we are able to version any element in the UML meta-model. Fig. 2 presents a simplified view of the relationship between the versioning model and the data model.

It is interesting to notice that our SCM approach has some features of Product Data Management (PDM) [15]: we use an object-oriented data model; our versioning model is not embedded into the data model (it points to elements in the data model); and we store the objects in a database, avoiding file system dependencies.

3.2. Homogeneous versioning

As discussed before, the software development process deals with different kinds of artifacts, such as use case descriptions, use case diagrams, class diagrams, sequence diagrams, code, test plans, test data, etc. All these artifacts must be controlled in a consistent way to provide snapshots of the system in different moments of development and maintenance. These snapshots, when applied to a formal revision, are called baselines or system configurations.

A system configuration can be seen as a version of the whole system. Each model element has its own version, but the aggregation of these model elements has another version: the configuration version. Thus, a configuration can be seen as a special kind of CI, a composite one [9]. It aggregates other CIs and its version is related to the versions of its parts. If a new version is created for some part of a composite CI, a new version should also be created for the whole CI. This strategy has already been applied by different file-based SCM systems [9].

In our approach both configurations and versions are dealt in the same way. If a CI is not composed of other CIs, the notion of version is the conventional one. However, if there is a composition relationship between CIs, the version of one CI depends on the version of the other. For example, a UML model has packages composed of classes and classes composed of attributes and operations. In this scenario, a package can be seen as a configuration of all its classes, and a class can be seen as a configuration of its attributes and operations. Assuming that attributes and operations are UV, if one attribute is changed, a new version of the class that encapsulates this attribute is also created, because the class has been indirectly changed. Due to the new version of the class, the package that contains this class will also receive a new version number. In other words, a configuration is defined at any level in the form of composite CIs in the UML model. The content of the configuration depends on the type of the composite CI, according to the UML meta-model. For instance, a package allows packages, classes, use cases, etc. as its contents. On the other hand, a class allows attributes and operations as its content.

This homogeneous way of treating configurations and versions allows future queries over a specific package or class and complete reconstruction of any previous state, with the correct set of attributes and operations. For instance, it is possible to ask for the most recent version of the root CI of a system (i.e. UML model). Due to the uniform metaphor for versions and configurations, it is easy to transform this element into a part of a bigger system because the whole system is seen by Odyssey-VCS as an ordinary CI. A possible drawback of this approach is the risk of an explosion of versions of composite CIs. However, file-based VCSs that use this approach deal with this problem by applying hard-links to sub-CIs that have not been changed.

3.3. Distributed and heterogeneous workspaces

The usage of a standardized data format is a key feature to support heterogeneous workspaces, maintained by different upper CASE tools. XMI is the most adopted format for both commercial and academic UML-based upper CASE tools. For this reason, Odyssey-VCS approach uses XMI as the format for communication between upper CASE tools and the SCM infrastructure. These tools can connect to Odyssey-VCS through the Internet and query for a specific version of a CI, modify it and send back to Odyssey-VCS.

3.4. Concurrent engineering

Odyssey-VCS is based on an optimistic strategy for concurrency control. The optimistic strategy, which has already been applied by file-based SCM systems since the 80s [9], lets developers change the same model in parallel, and merge the changes when the models are checked-in, as shown in Fig. 3. This strategy leverages parallel work, but increases the complexity of merge algorithms, as detailed later in Section 4.2.

The original configuration shown in Fig. 3(b) is the starting point of a new development cycle. The user configuration is created when a given developer checks-out the original configuration and performs some changes. During this period of time, other developers also work on the original configuration, merging their work into the current configuration. Finally, the user configuration is also merged with the current configuration, creating the final configuration.

When a conflict is detected during the merge procedure, the whole check-in is rolled-back and the developer receives a message containing a detailed conflict description and the original, user and current configurations. After performing a manual merge, which can be supported by external tools, the developer resubmits the UML model to

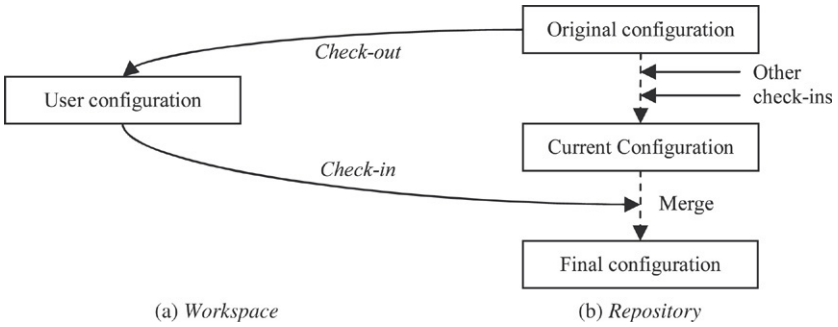

Fig. 3. Optimistic strategy for concurrency control.

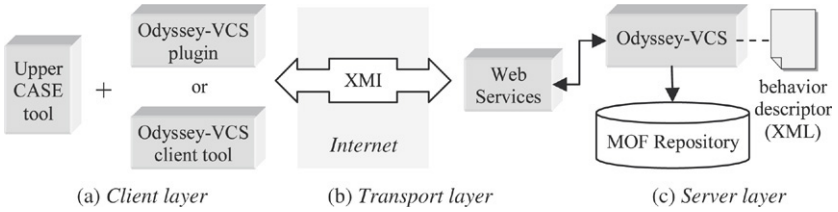

Fig. 4. Odyssey-VCS overall view.

the repository. Unfortunately, our approach only detects conflicts at check-in time, which may be too late. Some other research provides earlier awareness of concurrent work [45].

4. Odyssey-VCS internals

Odyssey-VCS was implemented in Java from scratch to avoid dependencies to existing file-based VCSs. However, it was built on top of a MOF Repository named MDR [33], reusing basic object persistence functionalities. Its architecture is composed of three major layers: client, transport, and server. The most important element of the *client layer*, presented in Fig. 4(a), is the upper CASE tool. We are assuming that this tool uses UML as the modeling notation and is able to externalize UML models using XMI. The integration between the upper CASE tool and Odyssey-VCS can be done via two alternative mechanisms: Odyssey-VCS plug-in and Odyssey-VCS client tool.

Some upper CASE tools offer an extension infrastructure that allows the addition of external tools. In this case, Odyssey-VCS plug-in can be used, providing a seamless integration. For instance, we adopted this mechanism to integrate Odyssey-VCS with the Odyssey environment [49]. However, some upper CASE tools have a poorly documented extension infrastructure, or do not even have it. In these situations, it is possible to use the Odyssey-VCS client tool. This tool opens an XMI file previously saved by the upper CASE tool and allows the execution of SCM commands, such as check-out and check-in. This mechanism was used to integrate Odyssey-VCS with Poseidon [5].

The *transport layer*, presented in Fig. 4(b), is responsible for allowing distributed development of UML models over the Internet. The current implementation of this layer uses local calls or Web Services [6] as a transport protocol. However, it can be replaced by other protocols, such as WebDAV, RMI, sockets, etc. Moreover, before being sent over the Internet, the XMI files pass through a compression layer (zlib) to increase the overall throughput of the transport layer. This kind of approach, instead of using deltas, is also being adopted by other SCM tools [16].

Finally, the *server layer* processes the XMI files, applying different versioning behaviors depending on the project needs. The checked-in XMI file is transformed into an object network, which is merged with existing objects and stored into MDR for further querying and retrieval. Currently, the server layer is centralized. The adoption of a distributed server layer, already applied by some file-based VCSs, such as ClearCase Multisite [50], BitKeeper [4], and GNU Arch [32], is left for future work. It is also important to notice that we do not version the XMI file directly. It is converted to an object-oriented representation by MDR and each object is individually versioned. The behavior configuration, the merge algorithm, and some versioning idiosyncrasies are presented in the next sections.

```
<type name="org.omg.uml.foundation.core.UmlClass">
        <UV>true</UV>
        <UC>true</UC>
</type>
<type name="org.omg.uml.foundation.core.Attribute">
        <UV>false</UV>
        <UC>true</UC>
</type>
<type name="org.omg.uml.foundation.core.Operation">
        <UV>false</UV>
        <UC>true</UC>
</type>
```

Fig. 5. Scrap of a behavior descriptor XML file.

Table 1
Odyssey-VCS merge algorithm

Case	$e \in O$	$e \in C$	$e \in U$	$e_O \equiv e_C$	$e_O \equiv e_U$	Action
1	T	T	T	T	T	Add e_C (or e_U) into F
2	T	T	T	T	F	Add e_U into F
3	T	T	T	F	T	Add e_C into F
4	T	T	T	F	F	Notify a conflict: "concurrent changes over the same element"
5	T	T	F	T	N/A	None (do not add "e" into F)
6	T	T	F	F	N/A	Notify a conflict: "concurrent removal and change over the same element"
7	T	F	T	N/A	T	None (do not add "e" into F)
8	T	F	T	N/A	F	Notify a conflict: "concurrent removal and change over the same element"
9	T	F	F	N/A	N/A	None (do not add "e" into F)
10	F	T	T	N/A	N/A	N/A
11	F	T	F	N/A	N/A	Add e_C into F
12	F	F	T	N/A	N/A	Add e_U into F
13	F	F	F	N/A	N/A	N/A

4.1. Behavior configuration

When a UML model element is checked-in, a specific action should be performed. However, as a flexible approach, Odyssey-VCS reads a behavior descriptor to decide what to do. This behavior descriptor informs which elements are the UC and the UV. For example, the scrap of a behavior descriptor shown in Fig. 5 indicates that no versioning information should be stored for attributes and operations. On the other hand, classes are considered as CIs (UV = true), meaning that every version should be registered. Moreover, classes are also considered atomic elements (UC = true). Due to that, Odyssey-VCS will notify a conflict when two or more people concomitantly check-in changes over the same class, even if the changes are on different parts of the class.

It is important to notice that every software development project has its own behavior descriptor. This allows the customization of Odyssey-VCS to the specific needs of projects. For instance, if a project does not define class as the UC, no conflict is detected when two people concomitantly check-in changes over different parts of it. On the other hand, if operation is set as the UV, every change in operations is registered together with versioning information.

Another important aspect is the interplay between UV and non-UV elements. Odyssey-VCS stores all physical versions of every element, but only stores logical versioning information of UV elements. For this reason, it is possible to correctly retrieve the context of a UV element, even if it is related to non-UV elements.

4.2. Merge algorithm

A built-in merge algorithm is also provided together with the flexible versioning infrastructure. This merge algorithm takes into account the configurations shown in Fig. 3. After analyzing the presence/absence of elements in these configurations, and the internal values of these elements after a check-in, we reached a complex scenario that is summarized in Table 1, which is inspired in classical merge algorithms [9]. The configurations and relations among them, used in Table 1, are defined as follows:

– O: Original configuration;
– U: User configuration;

– C: Current configuration;
– F: Final configuration;
– e_X: Element "e" in configuration "X"; and
– $e_X \equiv e_Y$: True if element "e" is identical in both configurations "X" and "Y".

Table 1 shows, for each possible scenario, which action should be taken by Odyssey-VCS. For example, case 3 shows a scenario where a given element (use case, for example) exists in all configurations ($e \in O, e \in C$, and $e \in U$), was changed in the current configuration ($e_O \not\equiv e_C$), but was not touched in the user configuration ($e_O \equiv e_U$). In this case, Odyssey-VCS promotes the element from the current configuration to the final configuration. On the other hand, case 6 shows a scenario where a given element (an operation, for example) was removed from the user configuration ($e \notin U$) but exists in all other configurations ($e \in O, e \in C$). However, the element was changed by other users ($e_O \equiv e_C$). As a result of this scenario, Odyssey-VCS notifies a conflict, arguing that the same element was removed and changed by different developers.

The results of the merge algorithm are consistent to the UML structure (guaranteed by the MDR repository), but may be inconsistent with UML well-formedness rules. The current release of Odyssey-VCS does not apply well-formedness rules consistency check. However, this validation can be obtained by external tools.

4.3. Versioning idiosyncrasies

Odyssey-VCS has some peculiarities regarding identification of UML model elements and relationships handling. Usually, the identification of configuration items is done via similarity analysis or explicit identifiers. When similarity analysis is in place, there is no need to modify the elements (i.e. it is non-intrusive) because it analyzes the element structure to detect that two elements are similar. However, it is ineffective when dealing with refactorings because it is not able to detect that both elements, before and after the refactoring, are the same. For this reason, we opted to use explicit identifiers in Odyssey-VCS. These explicit identifiers are attached to each checked-out model element in the form of a tagged value named Odyssey-VCS-ID. This tagged value stores the MOF id of the original model element.

This approach for model element identification has some important advantages. First of all, it is fast to find the original element of a given user element during check-in because the id of this original element is explicitly stored in a tagged value of the user element. The lookup for the original element runs in amortized logarithmic time due to the internal implementation of MDR repository, based on B-Tree data structure.

Moreover, the history of an element is kept even when this element is the subject of some kind of refactoring, such as renaming and moving. The use of explicit identifiers allows the detection of name and namespace changes, correctly establishing the version history besides the type of changes performed. For example, if a class named "Client" that resides in package "model" and already has 15 versions is moved to package "domain" and renamed to "User", the first version of class "User" would be 16, and the link to the 15 previous versions of this class would be available. This avoids a common incorrect behavior of some other VCSs: marking "Client" as deleted and considering "User" as a brand new CI, in version 1.

Finally, it is possible to create branches of a model element via simple copy and paste procedures. When a checked-out model element is copied and pasted into another part of a model, the identification of the original model element, which is stored in a tagged value attached to the element, is also pasted together with the pasted model element. This allows Odyssey-VCS to correctly keep the history of the copied and pasted elements during check-in.

However, there are also some drawbacks of using explicit identifiers. These drawbacks are related to the way ids are attached to the model elements. Currently, we are using the UML extensibility mechanism (i.e. tagged values) to perform this job. This solution is compatible to existing UML upper CASE tools, but makes it harder to extend Odyssey-VCS in the future to other MOF meta-models.

Another important aspect of Odyssey-VCS design is how relationships are handled. First, it is important to notice that relationships are elements defined in the UML meta-model. They inherit the *ModelElement* type and are dealt as any other UML model element. Due to that, we figured out two ways of handling relationships (a comprehensive discussion on relationship versioning is presented in the literature [51]): as individual elements or as primitive proprieties of existing elements. If relationships are seen as individual elements, both the UC and UV should be configured for each relationship and these elements should be handled by Odyssey-VCS in the same way as any other UML model element. However, if relationships are seen as primitive attributes of existing elements, there is

no need to configure the UC and UV for them, and the elements that are members of the relationships are marked as modified if the relationship changes. For example, if two classes, "Client" and "Product", are associated via a relationship "interested in", and this relationship is removed, changed (name, multiplicity, etc.), or other relationships are added, both classes would be marked as modified, because the relationship is seen as a primitive attribute of the classes.

5. Complementary work

The existence of a fine-grained model-based VCS, such as Odyssey-VCS, allows further research on improving this VCS and performing more sophisticated use of the versioned model-based repository. In our case, two complementary researches were performed over Odyssey-VCS. First, we built a CCS tightly integrated with Odyssey-VCS, to introduce change-based versioning capabilities in the Odyssey-SCM infrastructure. After that, we constructed a traceability link detection tool, which uses the integrated SCM repository to detect change traces among fine-grained model elements.

This section presents both complementary researches. First, we discuss how a change control process can be used together with Odyssey-VCS. Then, we introduce the concept of change traces and traceability link rationale. Finally, we describe how our traceability link detection tool is able to find dependencies among UML model elements using the integrated SCM infrastructure.

5.1. Change control process

The change control process (CCP) should be adapted to the different needs of specific organizations [25]. In addition, it is desirable to have automated systems that control the lifecycle of change requests through the CCP. Aiming to provide a flexible solution to this problem we built Odyssey-CCS, which is a CCS that provides CCP modeling capabilities following the SPEM notation [39] to the Odyssey-SCM infrastructure. Moreover, Odyssey-CCS also allows creating and assigning templates for each modeled activity. These templates are filled in during the activity execution.

It is possible to query Odyssey-CCS at anytime in the software development and maintenance lifecycle. Some examples of querying are: "what are the open change requests of a given project?" or "what is the quality assurance status of a given change?". Additionally, due to the integration of Odyssey-VCS and Odyssey-CCS in the same SCM infrastructure, it is possible to answer more complex queries, such as: "Who has approved to change this model element?" (information collected in the evaluation phase of the CCP) or "Why this model element has been changed?" (information collected in the change request phase of the CCP). Section 5.2 presents even more complex queries that can be answered by our integrated SCM infrastructure.

5.2. Change traces

When a software engineer changes a UML model, one of the most important questions he or she needs to answer is "Are there other elements that may be affected by these changes?". One way to assist software engineers in answering this question is to find traceability links between existing UML model elements, which indicate other elements that will probably need to be changed.

Change traces are defined as traceability links obtained from change information. Throughout the lifecycle of a project, software artifacts are changed when new features are added, old ones are enhanced or bugs are fixed. In our case, all these change information are stored in Odyssey-VCS and Odyssey-CCS repositories.

In the context of Odyssey-CCS, changes have a predefined lifecycle, determined by the modeled CCP, such as [24]: (1) change request, (2) classification, (3) impact analysis, (4) evaluation, (5) implementation, (6) verification, and (7) integration. During the implementation phase of this lifecycle, different software artifacts are modified to perform the change. Each version of the modified artifacts is stored in Odyssey-VCS. Along with the versioned artifacts, other valuable information is kept by the SCM infrastructure, such as change description, author, and date of each check-in operation performed during the implementation of a change.

The integration of information provided by Odyssey-VCS and Odyssey-CCS allows the detection of which artifacts are affected by a given change. Moreover, it offers an opportunity to gain some insight about change rationale [23].

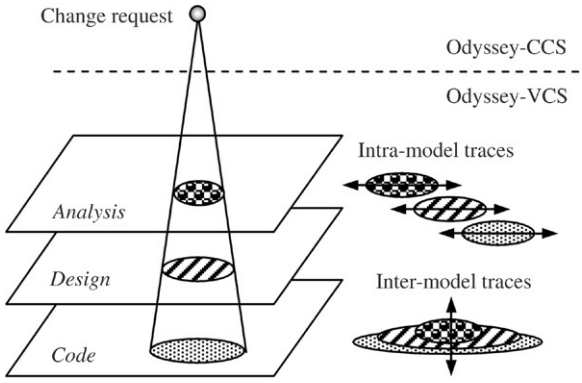

Fig. 6. Change traces.

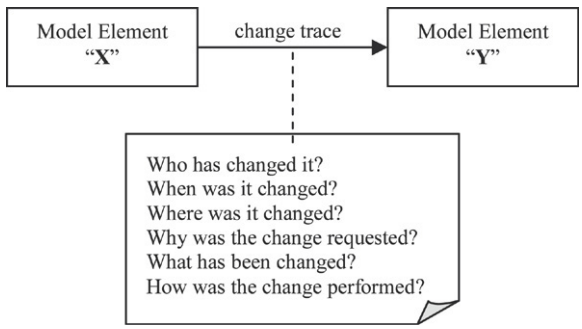

Fig. 7. Change traces rationale.

Fig. 6 shows two types of change traces that can be obtained by our approach: intra-model and inter-model. The intra-model change trace relates two or more elements at the same abstraction level. For example, it is possible to detect that two use cases are always changed together. On the other hand, inter-model change trace relates two or more elements at different abstraction levels. For example, it is possible to detect that a use case and a class are always changed together. It is important to notice that the grain of the model elements related by change traces depends on the UV configuration at Odyssey-VCS. Every UV model element is able to have change traces.

5.3. Traceability link rationale

While developers investigate change traces, they pose various questions to uncover the rationale of dependencies that were identified. Currently, most approaches that support traceability link detection fails to provide the semantics of the relations among software artifacts. To minimize this problem, our approach provides a view of all related artifacts that shall be updated and the details about the rationale, the history, and the people behind the change traces. Such details are vital in assisting developers to understand the current state of the software system throughout the evolution process [23].

This rationale follows an information structure widely used in the Computer Supported Cooperative Work literature to contextualize and provide knowledge about elements [11]. This information structure, shown in Fig. 7, is known as 5W+1H and comprises the following pieces of information: who, when, where, why, what, and how.

For example, the knowledge of who has performed a change may assist developers to understand if the change trace was introduced by a senior developer. Moreover, even if the change trace were introduced by a senior developer, other questions may indicate that this change trace was introduced just to fix a critical bug in the few days/hours before a release, lacking adherence to some coding standards used in the company.

However, in the real world of software development, with tight schedules and short time to market, manually recording such information is neither possible nor practical. On the other hand, SCM repositories store change details obtained from the SCM process. These change details are automatically collected and organized by our traceability link detection tool from the integrated SCM repository to provide the required 5W+1H information.

Table 2
Rationale gathering

Information type	Collection place
Who	• Check-in information from Odyssey-VCS • Implementation activity from Odyssey-CCS
When	• Check-in information from Odyssey-VCS • Implementation activity from Odyssey-CCS
How	• Impact analysis from Odyssey-CCS
Why	• Change request activity from Odyssey-CCS
What	• Check-in information from Odyssey-VCS • Verification activity from Odyssey-CCS
Where	• Check-in information from Odyssey-VCS

5.4. Change traces detection

The process of traceability link detection comprises two main phases: configuration and querying. The configuration phase is when the configuration manager sets the desired threshold values for some data mining metrics, as discussed later on, and informs the activities and fields of Odyssey-CCS that should be used to collect rationale information. The querying phase occurs when developers want to know the change traces among UML model elements. Usually, a UML model element is queried and all change traces for this element are provided. This mechanism helps to answer questions such as "Developers that change a given element also change which other elements?" providing support for the developers' future changes.

It is important to notice that this approach does not intend to replace the work of software engineers. Since change traces are based on past experience, they do not constitute absolute truth, but suggestions. To categorize the relevance of these suggestions, each change trace has a probabilistic interpretation based on the amount of evidence from the data that they are derived from.

This evidence is represented by two data mining metrics: support and confidence. These metrics are used to mine only frequent rules in databases via association rules techniques. Support quantifies the significance of the (co-)occurrence of artifacts in implemented changes. Confidence represents how much one artifact depends on others.

Typically, association rules techniques are interested in rules that satisfy both minimum support and confidence thresholds. In our approach, such thresholds should be set by the configuration manager at the configuration phase.

The process for mining change traces related to the queried model element comprises two steps: (1) selection of changes that will be analyzed, and (2) mining the changes, searching for rules that describe relationships between UML model elements.

Only changes that satisfy the following conditions are analyzed by the data mining algorithm: (1) the change should be completely implemented, (2) the change should be older than the UML model element that is being queried by the developer, and (3) the change should affect the UML model element that is being queried by the developer. The condition 1 is important to avoid detecting incorrect change traces due to incomplete changes. The condition 2 is important to focus only on the history of an element, not considering changes that impact newer versions of this element (it occurs only when the queried element is not part of the most recent configuration in the repository). Finally, the condition 3 is an optimization, to reduce the changes to be processed, allowing faster detection of change traces.

After that, our approach can apply data mining over the changes selected according to the above conditions. The data mining algorithm, based on Apriori algorithm [1], calculates the support and confidence metrics for each pair of artifacts that are included in the changes and, then, prunes the elements with support and confidence lower than the minimum thresholds.

Our approach presents, together with the mined traceability links, the rationale behind each change trace, as shown in Fig. 7. The rationale is composed of the 5W+1H information (shown in Table 2) collected from each change that participates in the mining process.

The automatic gathering of 5W+1H information depends on an integrated view of both Odyssey-VCS and Odyssey-CCS repositories. This integrated SCM repository is analyzed to collect the rationale information shown in Table 2. For each required piece of information, the gathering place should be provided in terms of the template and field that

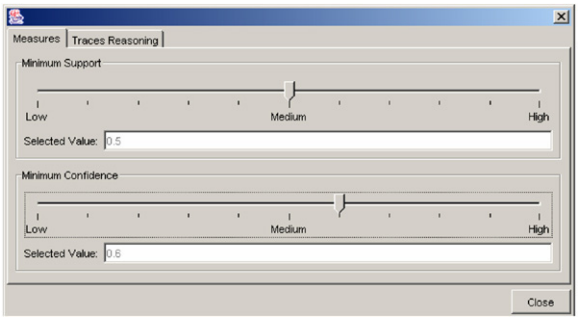

Fig. 8. Support and confidence configuration.

Table 3
Odyssey-VCS configuration

Element	Unit of Versioning (UV)	Unit of Comparison (UC)
Model	True	False
Package	True	False
Class	True	**True**
Attribute	True	True
Operation	True	True
Use Case	True	True
Actor	**False**	False

collects the desired piece of information in Odyssey-CCS, or directly collected in the Odyssey-VCS. For example, to collect the "why" information using a conventional CCP [24], the field "change description" in the template "change request" should be inspected.

6. Overall example

In this section we present an intentionally simple usage example that aims to illustrate how our approach provides flexibility and concurrent access during the evolution of UML models of a hotel network system. The next sections present the configuration phase of Odyssey-SCM, the grounding details of the hotel network system, a conflict handling scenario and a change trace detection scenario.

6.1. Odyssey-SCM configuration

Initially, suppose that Odyssey-VCS was configured as described in Table 3. It is important to notice that *Class*, a composite element that encloses *Attributes* and *Operations*, is configured as the UC. In other words, this means that should two or more developers check in changes concomitantly over the same class, a conflict will happen. This conflict occurs even if they are working over different parts of the class. Moreover, *Actor* is not configured as the UV. This means that no versioning information will be stored regarding this element.

Moreover, as discussed before, it is possible to select threshold values for support and confidence metrics in the traceability link detection tool, as shown in Fig. 8. By convention, support and confidence values are defined in a continuous domain with minimum and maximum boundaries equal to 0 and 1 (i.e. 0% and 100%), respectively. These values should be chosen by a configuration manager who knows the project needs and is capable to decide the best combination of support and confidence for this project. High values of support and confidence tend to detect fewer change traces. On the other hand, low values of support and confidence provide a huge amount of change traces, with many false positives.

We could detect that support and confidence should start with high values for new projects, and have their values decreased to a stationary position. A new project usually has a small SCM database. The number of elements in the database is an important factor in the computation of support and confidence metrics. In a small database, each element has a huge influence in the support and confidence values, increasing the number of false positives change

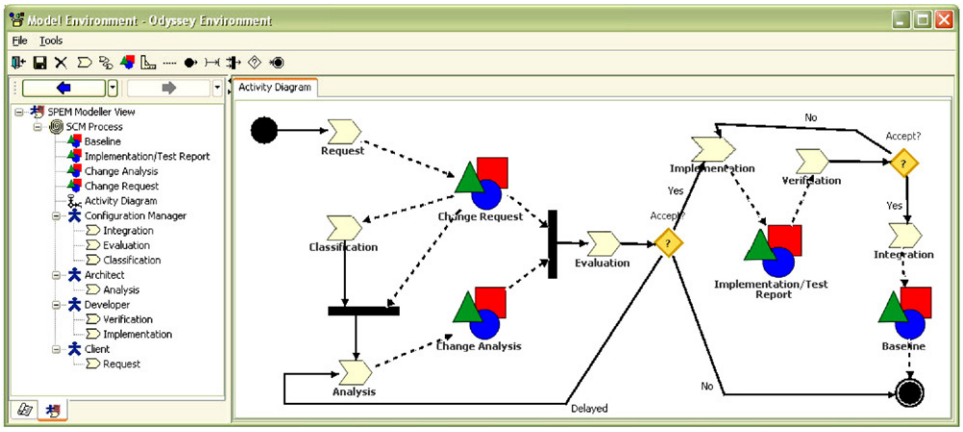

Fig. 9. Change control process modeled inside Odyssey-CCS.

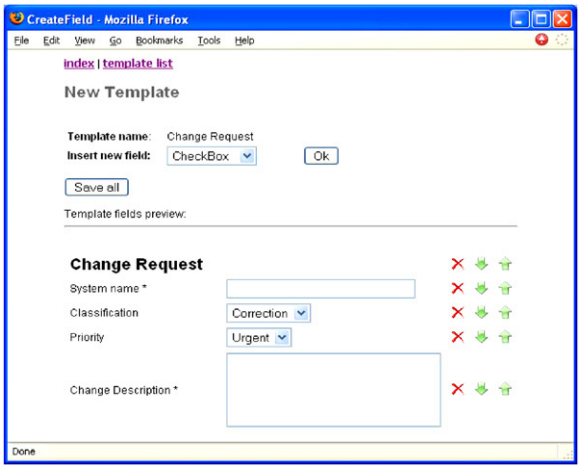

Fig. 10. Change request template being modeled using Odyssey-CCS.

traces. On the other hand, in a large database, an individual element has almost no influence on the value of support and confidence, allowing the reduction of support and confidence thresholds. Some other strategies for choosing support and confidence values are presented in the literature [53].

In this usage example, the data mining metrics have support and confidence set to 50% and 60%, respectively. The meaning of these metrics is that the queried artifact has been changed together with the traced artifacts in at least 50% of all existing changes, and the traced artifacts have been changed in at least 60% of the changes that affect the queried artifact. Therefore, if the configuration manager chooses a higher support, only artifacts modified together with the queried artifact in a higher number of changes will be returned.

Moreover, Fig. 9 presents the CCP modeled inside Odyssey-CCS to guide this example. This sample process, which follows the IEEE 828 standard [24], determines the activities and the products produced by these activities.

Together with this process, a set of templates are associated to the activity products. For example, the template that is associated to "Change Request", modeled in Fig. 10, will be presented to the client when a new change is being requested. The information filled in this template will be linked to all other information regarding the change and will be available for future queries, such as those performed by the traceability link detection tool.

The configuration manager, who is responsible for defining the CCP, establishes, for each question of the 5W+1H structure, where the traceability link detection tool should collect the pieces of information from the Odyssey-CCS repository, as shown in Fig. 11. The collected information will be organized to be presented to the software developer together with the change traces.

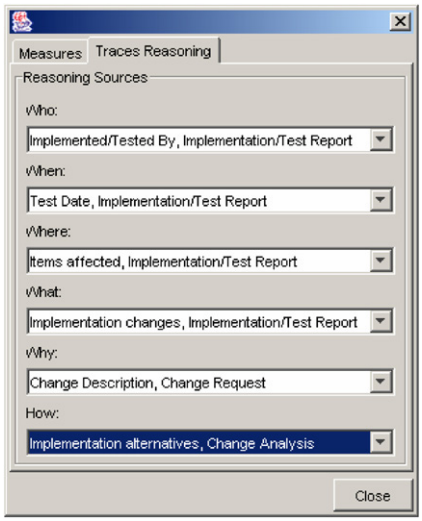

Fig. 11. Configuration of traceability link rationale gathering.

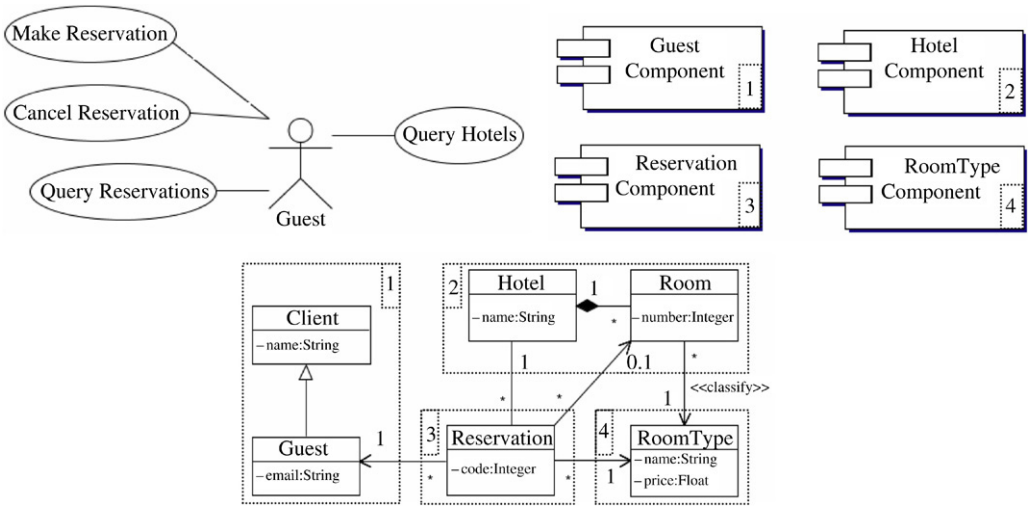

Fig. 12. UML model of the hotel network control system.

6.2. *Hotel network system*

The hotel network system is composed of four components, four use cases and six classes. Fig. 12 shows how components and classes in this example are related. By analyzing this figure, we can conclude that the "Guest" component is implemented by the "Client" and the "Guest" classes; the "Hotel" component is implemented by the "Hotel" and the "Room" classes; the "Reservation" component is implemented by the "Reservation" class; and the "RoomType" component is implemented by the "RoomType" class.

Initially, let us assume that the models presented in Fig. 12 were placed under version control using Odyssey-VCS. Moreover, let us assume that six changes were requested in Odyssey-CCS. These change requests, which intend to provide some basic required features for the hotel network control system, are listed in Table 4.

During the implementation phase of the changes lifecycle, some artifacts are created or modified using Odyssey-VCS. Table 5 shows all created or modified artifacts to accomplish the implementation of each change listed in Table 4. These artifacts are shown together with their final versions. It is worth mentioning that the relationship among change requests and changed artifacts is possible due to the integration of Odyssey-VCS and Odyssey-CCS repositories in the Odyssey-SCM infrastructure. Moreover, it is also important to notice that a change may be implemented by more than one check-in in Odyssey-VCS.

Table 4
Change requests

Change	Description
#1	Verify, during the reservation, the availability of rooms for the desired period.
#2	Show the room type, features, and price when the reservation is queried.
#3	Search for rooms in other hotels in the same area if there is no room available.
#4	Do not allow double reservations on the same room at the same period.
#5	Make the room available if the reservation is canceled.
#6	Show previously reserved hotels as preferred options.

Table 5
Modified artifacts

Change	Modified artifacts
#1	Use Cases: Make Reservation (v1), Modify Reservation (v2). Classes: Client (v2), RoomType (v3), Room (v1), Hotel (v1), Reservation (v1), and Guest (v3). Components: Reservation (v3), Hotel (v1), Guest (v3) and RoomType (v3).
#2	Use Cases: Query Reservation (v4). Classes: RoomType (v4) and Reservation (v4). Components: Reservation (v4) and RoomType (v4).
#3	Use Cases: Make Reservation (v5). Classes: Hotel (v5) and Reservation (v5). Components: Reservation (v5) and Hotel (v5).
#4	Use Cases: Make Reservation (v6). Classes: Hotel (v6) and Reservation (v6). Components: Reservation (v6) and Hotel (v6).
#5	Use Cases: Cancel Reservation (v7). Classes: Hotel (v8) and Reservation (v7). Components: Reservation (v7) and Hotel (v8).
#6	Use Cases: Query Hotels (v9). Classes: Hotel (v9) and Guest (v9). Components: Hotel (v9) and Guest (v9).

6.3. Conflict handling scenario

During the implementation of change number 1, described in Table 4, two developers, namely John and Mary, are using different upper CASE tools, respectively Poseidon and Odyssey, to concurrently change the initial version of the hotel network control system model. John wants to rename the *email* attribute of the *Guest* class to *telephone*, add the *gender* attribute into the *Client* class, and add a new use case named *Modify Reservation*. On the other hand, Mary wants to change the type of the *price* attribute of the *RoomType* class from *Float* to *Currency* and include two new operations in the *Guest* class: *getEmail():String* and *setEmail(email:String):void*.

The changes performed by John, shown in Fig. 13(a), were the first to be committed into the repository. No conflicts were detected and the commit was successfully merged into the current version of the repository. After John's commit, the current version of *Guest* class in the repository does not have the *email* attribute anymore. However, the version of the *Guest* class in the Mary workspace is out-of-date, still containing the *email* attribute, as shown in Fig. 13(b).

When Mary tries to commit her version, all UML model elements, except *Guest* class, are correctly merged. Even the sub-elements of the *Guest* class were individually merged successfully. However, besides their syntactical correctness, they are semantically incompatible, because the operations *getEmail():String* and *setEmail(email:String):void* were created to manipulate the *email* attribute, which was renamed *telephone* by John. Fortunately, *Class* type was defined as the UC. Due to that, a conflict is raised, as shown in Fig. 14.

Odyssey-VCS provides all necessary information to allow Mary fixing the conflict. This information comprises, in addition to Mary's local version of the model, the original and the current versions of the repository. After manually

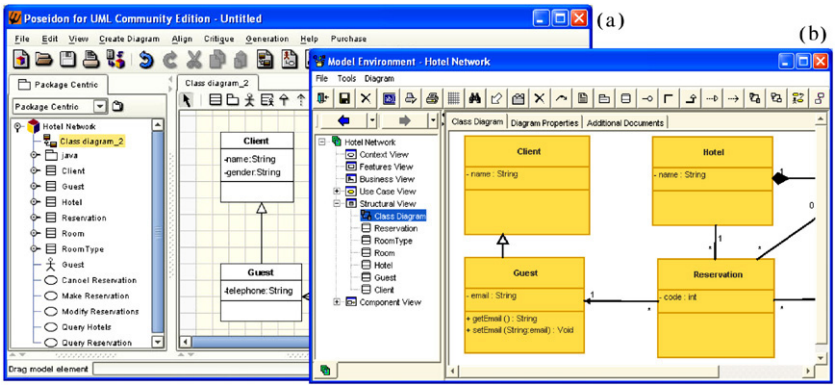

Fig. 13. Poseidon (a) and Odyssey (b) working over the same UML model.

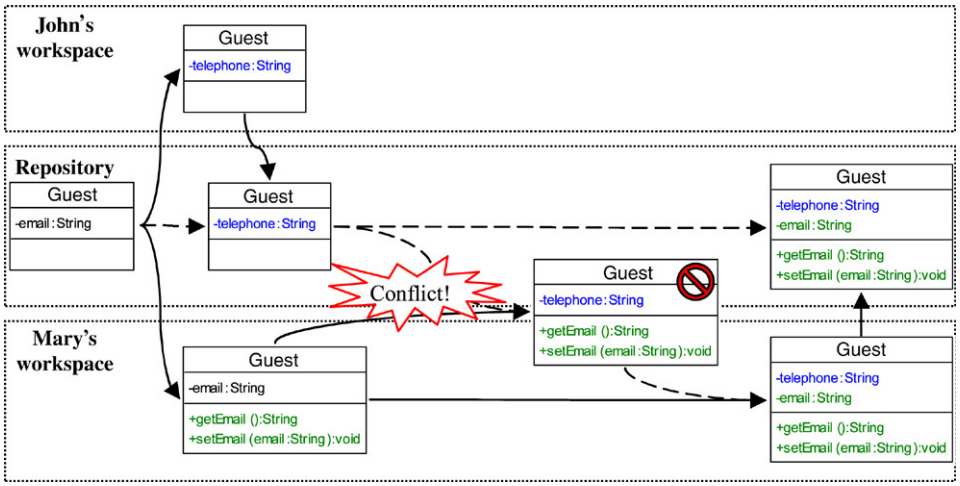

Fig. 14. Merge and conflict detection scenario.

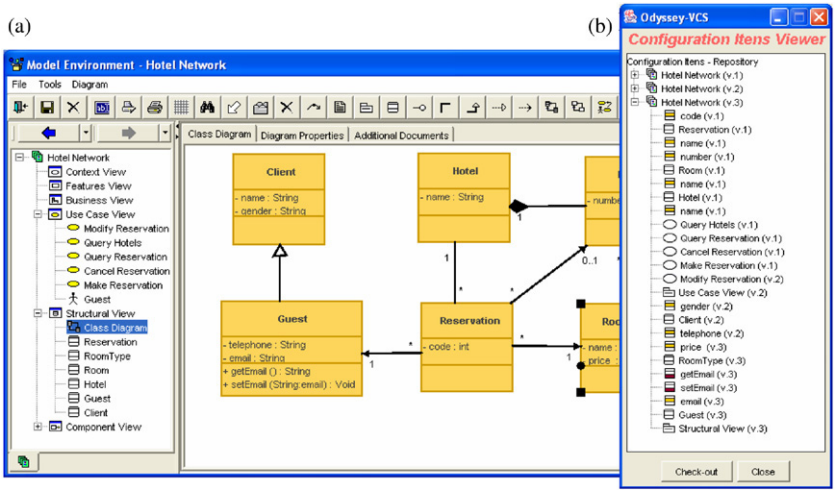

Fig. 15. Odyssey (a) performing check-out through Odyssey-VCS plug-in (b).

fixing the conflict, Mary is finally able to commit her changes into the repository. Fig. 15(b) shows the final state of the repository, which has the original version of the model, John's commit and Mary's commit.

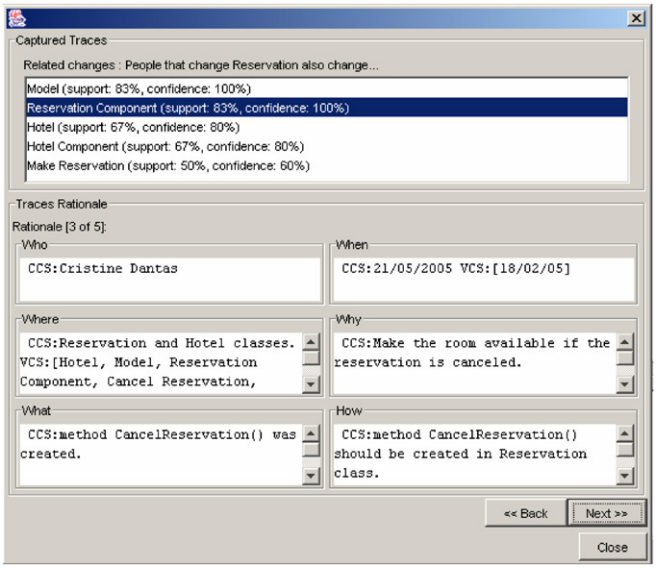

Fig. 16. Detected change traces.

Fig. 15(a) shows the version of the model in Mary's workspace after a new check-out. This version contemplates the original intention of both John and Mary. It is worth noticing that every version that is relative to a type defined as the UV in Table 3 has contextual information when stored in the repository. This contextual information, which encloses date, responsible, and additional comments, can be further used by other developers. However, *Actor* was not defined as the UV. For this reason, its versions are neither shown in Fig. 15 nor computed by Odyssey-VCS.

6.4. Change traces detection scenario

After all change requests presented in Table 4 were implemented, the developer received another change request (#7) and wants to query traceability links for a given model element. In this example, the queried model element is the "Reservation" class.

Besides the change trace to the model itself, which is always modified, the "Reservation" component is the most impacted artifact, with 100% confidence. Moreover, other traceability links are detected to the "Hotel" class and the "Hotel" component, with 80% confidence. Finally, the traceability link to the "Make Reservation" use case is detected with 60% confidence. Traceability links to other artifacts are also detected. However, the support or confidence is lower than 50% and 60%, respectively. These results are shown in Fig. 16.

At a first glance, these traceability links can be expected by the developer. However, in a real usage scenario, the number of changes implemented in the system is certainly greater than 6, as presented in this small example to clarify the mechanisms behind our approach. With a huge amount of UML model elements being changed, our approach becomes more relevant, detecting traceability links that would be hard to detect by manual approaches.

Fig. 16 also presents the rationale behind the traceability links. The "who" field was assigned to the person responsible for implementing the previous changes. This person is capable of answering questions about future implementations. The "when" field has the implementation date of the previous changes. This information gives an idea about how recent the previous implementations are. The "where" field defines the artifacts impacted by the previous changes. The "what" field explains what was done to implement the previous changes. The "why" field is related to the reason of the previous changes. Finally, the "how" field is related to the implementation strategies of the previous changes.

It is worth noting that some information, such as "who", "when", and "where", are provided simultaneously by Odyssey-VCS and Odyssey-CCS. Therefore, these pieces of information will be shown to the developer even if no information is filled in Odyssey-CCS. This occurs because the Odyssey-VCS automatically registers some versioning information, such as modified artifacts, author and date of check-in operation.

After the execution of this small usage example we conclude that this data mining approach can contribute to the impact analysis, avoiding under-prediction of the scope of a change and helping to identify critical side-effects.

Also, it can indicate anomalies in the design structure which may be subject to restructuring. For example, if the class "Reservation" of "Reservation" component is always changed together with a class of the "Hotel" component, then the architecture design probably has inconsistencies, due to the high coupling between these two classes.

In this example, change traces show which artifacts have been changed together with the "Reservation" class. Based on this information, the developer can decide if the mined artifacts will be changed together or not. The rationale provided together with the change traces is an important tool to help in this decision.

However, significant refactorings of the analysis and design models can negatively affect our approach in the same manner it can negatively affect other traceability link detection approaches based on historical modifications. This occurs because the historical information is kept in Odyssey-VCS even when artifacts are removed or renamed. As a result, some traceability links may be incorrectly detected. Fortunately, after the implementation of new changes, the approach will naturally return to a consistent state due to the characteristics of data mining algorithms.

This section presented a simple example of Odyssey-SCM usage. This example motivates a more formal usability evaluation of Odyssey-SCM. However, Odyssey-SCM is still an academic prototype and would need additional effort to be transformed into a product prior to any usability evaluation. Due to that, in the next section we focus on the evaluation of aspects regarding efficiency, such as performance and scalability. The evaluation of other important quality characteristics [27], such as functionality, reliability, usability, maintainability, and portability, is left for future work.

7. Evaluation

The necessity of SCM systems for high-level artifacts, such as UML models, is widely highlighted in the literature [7,9,16,24,35]. However, an important requirement for such systems is related to scalability [14]. For this reason, we executed a study to characterize Odyssey-VCS in terms of the existing file-based SCM systems. Our goal is to understand how difficult it is to scale when versioning UML model elements if compared to traditional file-based versioning. This study was performed over historical data of two existing systems. The first system is a small tool for importing and exporting UML models in XMI format (around 2000 physical lines of Java code). The second system, named Odyssey [49], is a medium size software development environment being developed at COPPE/UFRJ since 1997 (around 60,000 physical lines of Java code). We recognize these systems as small systems in terms of SCM. However, they are big enough to show the differences among file-based and model-based SCM. We will call these systems, respectively, S1 and S2 from now on.

To perform the study, we gathered S1 versioning data produced during the period of October 24, 2003 and September 14, 2005 and S2 versioning data produced during the period of November 26, 2003 until December 11, 2003. We reorganized the data to replicate the original check-outs and check-ins that took place, and then replayed those check-outs and check-ins anew into the CVS, Subversion, and Odyssey-VCS repositories. In the case of Odyssey-VCS, we performed a reverse engineering step before each check-in to obtain the corresponding UML model. The result was that, during the playback, we could reproduce the original scenario of development and maintenance. This strategy made it possible to look back in time and collect performance metrics of the three systems in action.

The next sections detail our planning of the retrospective study, our preparation of the environment for the study, the statistics that were gathered, the execution of the study, and the subsequent analysis of the results that we obtained. The result is dual: (1) a demonstration of Odyssey-VCS in action, illustrating that it is able to effectively version UML models, and (2) a detailed understanding of where Odyssey-VCS approach can be improved to achieve better scalability.

7.1. Study planning

This evaluation consists of an observational study via an "in vitro" benchmark to characterize Odyssey-VCS in terms of the existing file-based SCM systems [28,29,52]. We want to observe Odyssey-VCS in a controlled environment, but using real data, and detect the weakness of our approach and some generic challenges of model-based versioning. In our study scenario, we have two independent variables (factors): SCM system and software development project. The treatment factors of the SCM system variable are CVS, Subversion, and Odyssey-VCS. On the other hand, the treatment factors of the software development project variable are S1 and S2. As discussed before, S1 and S2 are completely different systems in terms of complexity and size.

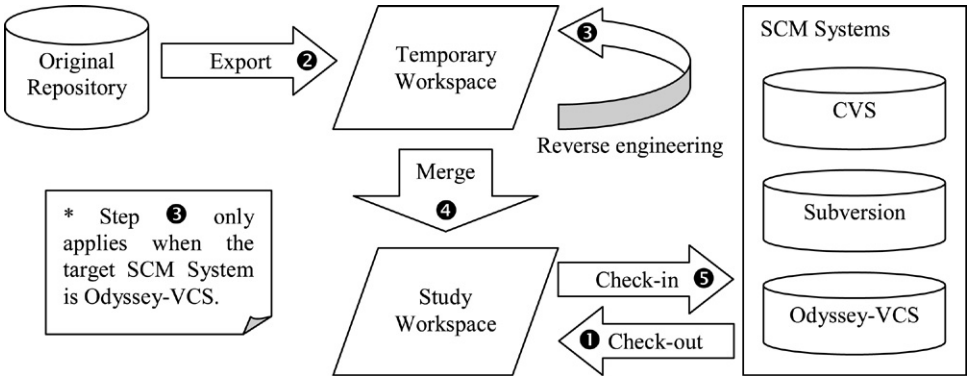

Fig. 17. Retrospective study overview.

Although CVS and Subversion are file-based SCM systems, they are widely used in commercial and open-source projects. For this reason, we decided to use them as control objects (baselines) of our study. We do not assume that version control over file-based artifacts has the same complexity of version control over model-based artifacts. However, we want to know how different these two scenarios are for a given software development project.

Our main dependent variables are the check-out and check-in duration. Moreover, we observed the number of CIs, the workspace size, and the repository sizes for each possible factor combination (<CVS, Subversion, Odyssey-VCS> x <S1, S2>). Finally, we also observed the amount of time spent in the transformation of XMI files into object-oriented structures that are manipulated by Odyssey-VCS and vice-versa. This transformation occurs during all check-in and check-out operations because Odyssey-VCS works over a MOF compliant object-oriented repository, the MDR.

7.2. Environment preparation

The study execution demanded two main components: (1) available and organized historical data, and (2) a playback tool. The historical data was obtained from an existing SCM system. This existing system, which is CVS, does not organize the data in a per commit fashion. Using Subversion it is possible to directly access a specific configuration using its global version identification. It occurs because a sequential version number is assigned for each commit. For this reason, we used cvs2svn tool to transform the existing CVS repository into a Subversion repository.

Having both S1 and S2 historical data ordered by commits, we developed a tool that is able to export each existing configuration and check them into a monitored repository. The overview of the process implemented by this tool is presented in Fig. 17. This process, which is composed of five steps, is repeated for each configuration that is analyzed in the study (126 configurations for S1 and 52 configurations for S2).

The state of the environment before the start of the study is the following: The original repository contains all versions of S1 or S2; both "Temporary Workspace" and "Study Workspace" are empty; and the target "SCM System" repository is also empty.

The first step (❶) populates the "Study workspace". This step is performed for each iteration, except the first one. This occurs by checking-out the last configuration stored in the "SCM System" under study (CVS, Subversion, or Odyssey-VCS). Both CVS and Subversion construct a file/directory structure on the "Study Workspace". However, the result provided by Odyssey-VCS is a single XMI file holding the checked-out model. The "Study Workspace" will remain empty if no configuration has been committed to the "SCM System" (first iteration).

In the second step (❷) a configuration is exported from the "Original Repository", which contains S1 or S2, into the "Temporary Workspace". If the "SCM System" under study is Odyssey-VCS, the third step (❸) should be performed. This step applies reverse engineering over the "Temporary Workspace", obtaining the UML class model from the source code. This step is performed using two tools developed in the context of the Odyssey project: Ares (a reverse engineering tool) and Odyssey-XMI (a tool for exporting/importing XMI).

At this moment, both "Temporary Workspace" and "Study Workspace" store artifacts at the same abstraction level: files and directories in the case of CVS and Subversion, and XMI UML model in the case of Odyssey-VCS.

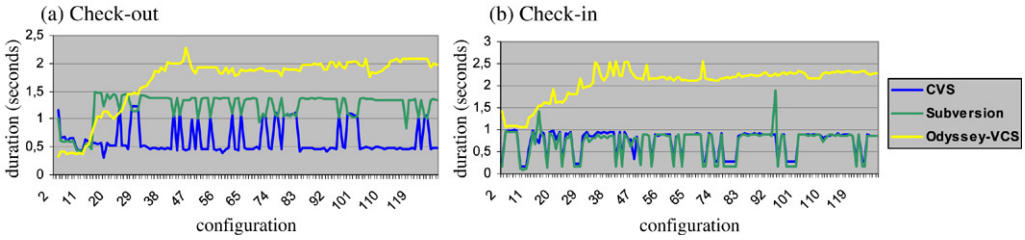

Fig. 18. S1 check-in and check-out results.

However, the "Temporary Workspace" stores plain artifacts at version N, and "Study Workspace" stores versioned artifacts at version $N-1$. These versioned artifacts are composed of the original artifacts at version $N-1$, together with versioning information (CVS directory, .svn directory or tagged values, for CVS, Subversion or Odyssey-VCS, respectively). The fourth step (❹) consists in merging plain artifacts at version N, located at "Temporary Workspace", with already versioned artifacts at version $N-1$, located at "Study Workspace". This merge algorithm is more complex than a simple copy, because both CVS and Subversion should be notified if a new element is added or removed from the workspace ("cvs add"/ "cvs remove" or "svn add"/ "svn delete" commands).

Finally, the fifth step (❺) checks-in the "Study Workspace" into the "SCM system" under study. After this step, both workspaces are emptied and the whole process restarts for the next configuration in the "Original Repository".

7.3. Statistics gathering

This retrospective study aims to analyze different statistics gathered from CVS, Subversion, and Odyssey-VCS execution. To allow this automatic gathering, we instrumented our playback tool presented in Fig. 17. Moreover, we also instrumented Odyssey-VCS tool to collect statistics regarding XMI parsing and serialization. A total of 20 metrics were collected from each processed configuration of S1 and S2. These metrics are: the configuration number, author, and date; the workspace size, the number of CIs in the workspace, the repository size, the check-in duration, and the check-out duration for each analyzed SCM system; and the XMI parsing and serialization duration for Odyssey-VCS system.

7.4. Study execution

The study was executed in a Pentium IV 2.4 GHz Hyper-threading, equipped with 1 GB RAM and 80 GB RAID 1 composed of two 80 GB serial ATA disks, running Windows XP and Cygwin. Special attention was given to diminish external interferences, such as antivirus and other daemon processes that could affect the final results. We also decided to execute the study in a local environment to focus on the SCM systems versioning performances. For this reason, the collected statistics do not take into account network communication delays.

Odyssey-VCS behavior descriptor has been configured to consider packages, classes, methods, and attributes as the UV and classes, methods, and attributes as the UC. The raw results of the study were stored by our playback tool into comma separated values (CSV) files for further analysis. These files were processed using MS Excel tool.

7.5. Data analysis

Aiming to characterize Odyssey-VCS, we first analyzed check-out and check-in performances over system S1, as shown in Fig. 18. In both check-out and check-in cases we noticed that Odyssey-VCS has not performed as well as CVS and Subversion. Besides the differences, it is important to notice that the duration of check-out and check-in is not affected by the number of configurations previously checked-in the repository. This situation depicts scalability in project length dimension. For instance, the check-out or check-in duration is around 2 s for any configuration from 38 up to 127, which represents almost 2 years of development. However, this situation raises an important question: "Does it also scale in project size?".

To answer this question we performed the same analysis over S2, which is much bigger than S1. The results presented in Fig. 19 confirm the scalability in project length dimension because the check-out and check-in duration

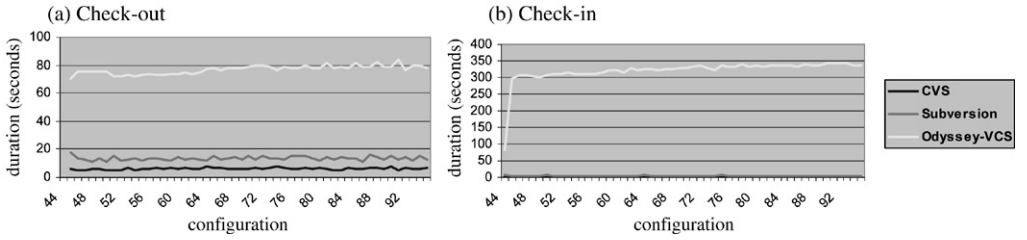

Fig. 19. S2 check-in and check-out results.

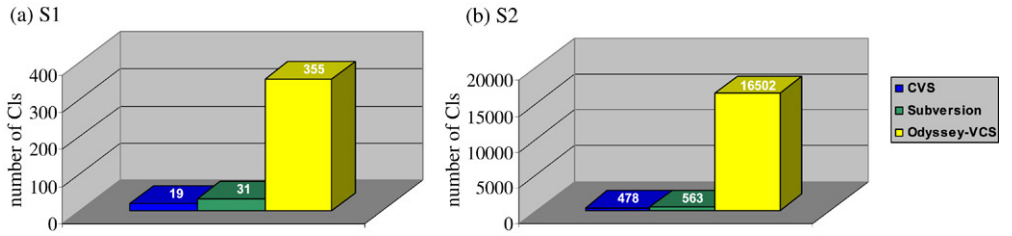

Fig. 20. Number of configuration items.

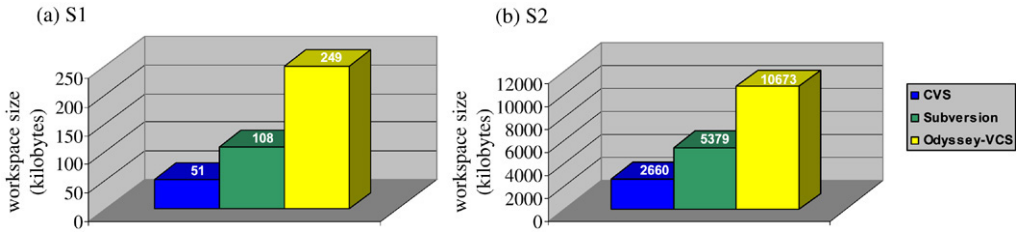

Fig. 21. Workspace size.

are still not affected by the number of existing configurations in the repository. But Fig. 19 also shows that it does not scale in project size dimension.

This second analysis brought other questions: "Why Odyssey-VCS does not scale up in project size dimension?" "Is it a limitation of Odyssey-VCS or is it related to the underlying technology (MOF, MDR, XMI, etc.)?". To answer these questions, we performed three further understanding analysis: (1) comparison of the number of configuration items in CVS, Subversion, and Odyssey-VCS workspaces; (2) analysis of the workspace size of CVS, Subversion, and Odyssey-VCS; and (3) internal analysis of Odyssey-VCS, trying to figure out the responsibility of MDR in the total check-out and check-in duration.

The first understanding analysis shows the version explosion phenomenon for fine-grained elements. When analyzing S1, we can notice that the average number of CIs processed by Odyssey-VCS is considerably larger than CVS or Subversion. CVS considers only files (classes) as CIs. On the other hand, Subversion considers directories (packages) in addition to files as CIs. However, Odyssey-VCS was considering packages, classes, methods, and attributes as CIs. Due to that, Odyssey-VCS has processed 19 times more CIs than CVS and 11 times more CIs than Subversion. These results are even more impressive when S2 is analyzed: Odyssey-VCS has processed 34 times more CIs than CVS and 29 times more CIs than Subversion, as shown in Fig. 20. It is worth noticing that the number of versions of CIs is even greater, because one CI usually has more than one version associated with it.

The second understanding analysis is focused on the average size of the workspace that is being checked-out and checked-in. As discussed before, the Odyssey-VCS workspace is constructed by reverse engineering the source code. For this reason, all information presented in the UML model is derived from the source code. However, when analyzing the workspace size we could notice that the UML model is up to 5 times bigger than the respective source code, which includes the versioning information stored in CVS or .svn directories. This comparison is shown in Fig. 21.

Finally, the third understanding analysis was on the impact of transforming such big XMI files, which are present in the Odyssey-VCS workspace, into object-oriented networks that are stored in the Odyssey-VCS repository. The

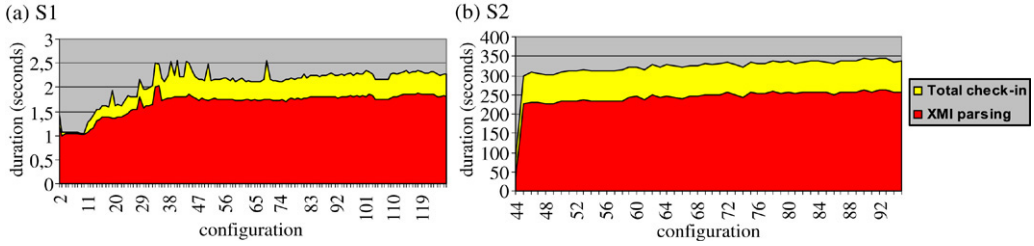

Fig. 22. XMI parsing during check-in.

parsing and serialization of XMI is performed by MDR. During check-out, the selected version is immediately accessed via B-Tree indexes and transformed into XMI. For this reason, 100% of the check-out time is devoted to XMI serialization. During check-in, the XMI is transformed into an object-oriented network and processed by Odyssey-VCS. In average, from 75% up to 81% of the check-in time is devoted to XMI parsing, as shown in Fig. 22.

7.6. Final remarks

This characterization study has analyzed Odyssey-VCS in terms of scalability. We considered two scalability dimensions: project length and project size. The project length dimension is focused on how the system behaves when the repository contains a large number of artifacts. On the other hand, the project size dimension is concerned with how the system behaves when the workspace contains a large number of artifacts.

The results showed that Odyssey-VCS performed well regarding the project length dimension. The current storage mechanism of Odyssey-VCS is MDR, which implements a B-Tree to physically store the versioned configuration items. B-Tree allows amortized logarithmic time for lookups, insertions, and deletions.

However, Odyssey-VCS did not scale-up regarding the project size dimension. After some additional analysis we could find some explanations to this problem. First, the number of configuration items is huge when versioning fine-grained UML model elements (up to 34 times bigger in some cases). Moreover, the XMI representation itself does not scale-up for representing huge systems. For instance, S2 has around 60,000 physical lines of Java code and its reverse engineered class model has around 10.5 MB. Finally, the parsing of huge XMI files is a very slow activity. MDR took around 241 s to read the 10.5 MB XMI file into the B-Tree during each check-in, and around 76 s to reconstruct the XMI file during each check-out.

Our final conclusion is that the current specifications for UML model exchange (XMI) and some implementations of MOF repository (i.e. MDR) are not designed to deal with huge systems. Complementary research should be done on alternative ways of exchanging UML models, such as default compression layer for XMI (we already use a zlib layer for that) and XMI diff (the XMI specification already defines diff tags, but most tools are not compliant to these tags). Another important research is on fast algorithms for parsing and serializing huge XMI files into object-oriented networks and further storage of these objects in database structures such as B-Tree.

8. Related work

The related work can be divided into two main points of view: (1) **SCM systems** themselves, and (2) **use of SCM repositories** to detect dependencies among software artifacts.

In the **SCM systems** point of view, most commercial and open-source SCM systems are based on file system data model [8,17,44,50]. As previously discussed, these SCM systems have several limitations to manipulate artifacts with complex internal data model. However, these solutions are generic and mature, being suitable for versioning text-based artifacts. We do not see these approaches as direct competitors of our approach. Our approach can be used to version UML model elements while these approaches can be used to version source code in the same software development project. Some upper CASE tools, such as Enterprise Architect [47] and Poseidon [20], provide integration with existing file-based VCS. However, they are subject to the problems discussed in Section 2.

There are also other approaches that employ other data models, such as entity-relationship or even object-oriented. However, these approaches work at source code level and are focused on a specific programming language. For instance, Goldstein et al. [21], Habermann et al. [22], and Render et al. [43] support versioning of Smalltalk, C and

Pascal source code, respectively. Our approach can also be seen as complementary to these approaches since we are focused on UML model elements and these approaches are focused on source code.

A few approaches use non-file system data model to version analysis and design artifacts. For instance, Ohst et al. [35] propose an approach for versioning analysis and design artifacts via syntax trees stored in XML files. Working at fine-grained UML artifacts they can correctly manage structural changes on these artifacts. However, the usage of XML does not mean adherence to modeling standards. Their XML format does not follow XMI specification for UML, leading to incompatibilities with existing UML-based upper CASE tools. Moreover, Nguyen et al. [34] use a hyper-versioning system to apply version control over complex artifacts, including UML analysis and design artifacts. This work has a strong focus on versioning relationships among the elements. However, it is also based on a proprietary UML data model, reducing compatibility with existing upper CASE tools.

Finally, OMG is working on a specification for MOF versioning [38]. Besides the non-existence of a completely final specification version up to this moment, it is possible to notice that Odyssey-VCS is pretty adherent to the specification philosophy. Similar to the specification, Odyssey-VCS has its own versioning meta-model. Moreover, it stores the versioned elements into separate per-version extents with associated history of changes.

In the **use of SCM repositories** point of view, researchers have tried to understand patterns of software evolution. Gall et al. [19] use release data to detect logical coupling between modules. Ball et al. [2] have performed some cluster analysis of C++ classes stored in SCM repositories.

Some recent works also perform historical analysis over SCM repositories. Shirabad et al. [46] use inductive learning to find out different concepts of relevance among logically coupled files. Eick et al. [13] argue that code decay is related to the difficulty to perform changes. For this reason, they analyze change history applying decay indexes to identify risk factors. Draheim et al. [12] argue that product quality is dependent of process quality. Due to this argument, the development process activities are analyzed and some metrics are applied over a VCS. Finally, Zimmermann et al. [53] have evidenced that mining SCM repositories can be useful for suggesting and predicting likely further changes, detecting hidden dependencies, and preventing errors due to incomplete changes.

However, these related researches work over file-based SCM repositories. For this reason, they lack support for traceability link detection of fine-grained UML model elements. Moreover, they do not provide change traces rationale, automatically extracted from an integrated SCM infrastructure, as we do in our work.

9. Conclusion

In this paper we presented a SCM approach for UML models. Our approach differs from existing approaches in the following aspects. First, we provide support for flexibility during CI identification, allowing the configuration of the UC and the UV for UML model elements. Second, our approach is based on well-adopted standards, such as UML, MOF, and XMI, raising the compatibility with existing upper CASE tools. We also have focused on current challenges of SCM to avoid reinventing the wheel regarding already solved problems. Finally, we allow configuration managers to model their own change control process and to assign templates to be filled in during the process execution. Besides, our approach also provides a built-in merge algorithm, supporting concurrent development.

Moreover, we presented a data mining approach that is able to detect dependency among UML model elements stored in a fine-grained SCM repository. This approach has the following features: (1) automatic change traces detection; (2) use of UML model elements as mining units, leveraging the state of the art of SCM data mining to fine-grained analysis and design artifacts; (3) description of change traces rationale using 5W+1H structure, automatically collected from an integrated SCM infrastructure; and (4) use of Web Services to allow remote access from different CASE tools. It is worth noting that we could not find another approach that applies data mining over UML model elements, nor automatically collects the rationale that support detected traceability links.

Our approach, however, is currently tightly coupled to the UML meta-model. At this moment, a project should apply in parallel Odyssey-SCM to version UML together with a file-based VCS to version other types of files. An important future work is the generalization to the MOF meta-model layer, allowing versioning of any MOF compliant meta-model (OMG is currently finalizing MOF meta-models for different languages, such as Java, EJB, COBOL, PL/1, C, and C++ [18]). Additional future work consists on allowing users to customize the merge algorithm for their specific needs and to keep information regarding teams that produce or reuse black-box components. This support is useful for deciding who is responsible for maintaining a given component in the future.

Other limitations of the Odyssey-SCM current release regards configuration constraints, partial check-outs, and workspace caching. Actually, many of these aspects are envisioned by our approach. However, for sake of simplicity, we adopted simpler solutions for the first release of the prototype.

Additionally, the current release of Odyssey-VCS does not use deltas to compose versions. On the one hand, the negative impact of this decision is higher network traffic. On the other hand, we do not need to compute a version based on prior versions and deltas, which saves some CPU cycles during check-out and check-in. We also use zlib to help reduce the transport overhead due to the absence of deltas. While performance and scalability were not a major concern for this first prototype, after an initial evaluation of Odyssey-VCS we could find some new research challenges regarding these issues for the next releases. After improving the performance and scalability of Odyssey-VCS, we intend to run some case studies in real software development scenarios.

Acknowledgements

Our thanks to the members of the Software Reuse Group at COPPE/UFRJ and to Márcio Barros, who helped in some discussions regarding Odyssey-VCS evaluation. Moreover, we would like to thank CNPq, CAPES, and Brazilian Central Bank for partially financing this project.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: International Conference on Very Large Data Bases, Morgan Kaufmann, Santiago de Chile, Chile, 1994, pp. 487–499.
[2] T. Ball, J. Kim, A.A. Porter, H.P. Siy, If your version control system could talk, in: Workshop on Process Modelling and Empirical Studies of Software Engineering, Boston, MA, 1997.
[3] S. Beydeda, M. Book, V. Gruhn, Model-Driven Software Development, Springer, 2005.
[4] BitMover, BitKeeper. http://www.bitkeeper.com, accessed in April 16, 2006.
[5] M. Boger, T. Sturm, E. Schildhauer, E. Graham, Poseidon for UML User Guide, Gentleware AG, 2000.
[6] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard, Web Services Architecture—W3C Working Group Note. http://www.w3.org/TR/ws-arch, accessed in April 16, 2006.
[7] M.B. Chrissis, M. Konrad, S. Shrum, CMMI: Guidelines for Process Integration and Product Improvement, Addison-Wesley, Boston, MA, 2003.
[8] B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato, Version Control with Subversion, O'Reilly, 2004.
[9] R. Conradi, B. Westfechtel, Version models for software configuration management, ACM Computing Surveys 30 (2) (1998) 232–282.
[10] C.R. Dantas, L.G.P. Murta, C.M.L. Werner, Consistent evolution of UML models by automatic detection of change traces, in: International Workshop on Principles of Software Evolution, IWPSE, Lisbon, Portugal, 2005, pp. 144–147.
[11] P. Dourish, V. Bellotti, Awareness and coordination in shared workspaces, in: Conference on Computer Supported Cooperative Work, ACM Press, Toronto, Canada, 1992, pp. 107–114.
[12] D. Draheim, L. Pekacki, Process-centric analytical processing of version control data, in: International Workshop on Principles of Software Evolution, Helsinki, Finland, 2003, pp. 131–136.
[13] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus, Does code decay? Assessing the evidence from change management data, IEEE Transactions on Software Engineering 27 (1) (2001) 1–12.
[14] J. Estublier, Software configuration management: A roadmap, in: International Conference on Software Engineering, The Future of Software Engineering, Limerick, Ireland, 2000, pp. 279–289.
[15] J. Estublier, J.-M. Favre, P. Morat, Toward SCM/PDM integration? in: Software Configuration Management, SCM, Springer Verlag, Brussels, Belgium, 1998, pp. 75–95.
[16] J. Estublier, D. Leblang, A.v.d. Hoek, R. Conradi, G. Clemm, W. Tichy, D. Wiborg-Weber, Impact of software engineering research on the practice of software configuration management, ACM Transactions on Software Engineering and Methodology 14 (4) (2005) 1–48.
[17] K. Fogel, M. Bar, Open Source Development with CVS, The Coriolis Group, Scottsdale, Arizona, 2001.
[18] D.S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, Inc., Indianapolis, Indiana, 2003.
[19] H. Gall, M. Jazayeri, R. Klösch, G. Trausmuth, Software evolution observations based on product release history, in: International Conference on Software Maintenance, Bari, Italy, 1997, pp. 160–196.
[20] Gentleware, Poseidon for UML. http://www.gentleware.com, accessed in April 16, 2006.
[21] I.P. Goldstein, D.G. Bobrow, A layered approach to software design, in: D.R. Barstow, H.E. Shrobe, E. Sandewall (Eds.), Interactive Programming Environments, McGraw-Hill, New York, 1984, pp. 387–413.
[22] A.N. Habermann, D. Notkin, Gandalf: Software development environments, Transactions on Software Engineering 12 (12) (1986) 1117–1127.
[23] A.E. Hassan, R.C. Holt, ADG: Annotated dependency graphs for software understanding, in: Visualizing Software For Understanding and Analysis, Amsterdam, Netherlands, 2003, pp. 41–45.
[24] IEEE, Std 828—IEEE Standard for Software Configuration Management Plans, Institute of Electrical and Electronics Engineers, 2005.
[25] IEEE, Std 1042—IEEE Guide to Software Configuration Management, Institute of Electrical and Electronics Engineers, 1987.
[26] ISO, ISO 10007, Quality Management—Guidelines for Configuration Management, International Organization for Standardization, 1995.

[27] ISO, ISO/IEC 9126 – Software Engineering – Product Quality, International Organization for Standardization, 2001.
[28] N. Juristo, A.M. Moreno, Basics of Software Engineering Experimentation, Kluwer Academic Publishers, 2001.
[29] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K.E. Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Transactions of Software Engineering 28 (8) (2002) 721–734.
[30] A. Leon, A Guide to Software Configuration Management, Artech House Publishers, Norwood, MA, 2000.
[31] L.G.B. Lopes, L.G.P. Murta, C.M.L. Werner, Odyssey-CCS: A change control system tailored to software reuse, in: International Conference on Software Reuse, Torino, Italy, 2006.
[32] T. Lord, GNU Arch. http://www.gnuarch.org, accessed in April 16, 2006.
[33] M. Matula, NetBeans Metadata Repository. http://mdr.netbeans.org, accessed in April 16, 2006.
[34] T.N. Nguyen, E.V. Munson, J.T. Boyland, The molhado hypertext versioning system, in: Conference on Hypertext and Hypermedia, Santa Cruz, USA, 2004, pp. 185–194.
[35] D. Ohst, U. Kelter, A fine-grained version and configuration model in analysis and design, in: International Conference on Software Maintenance, ICSM, Montreal, Canada, 2002, pp. 521–527.
[36] H.L.R. Oliveira, L.G.P. Murta, C.M.L. Werner, Odyssey-VCS: A flexible version control system for UML model elements, in: International Workshop on Software Configuration Management, Lisbon, Portugal, 2005, pp. 1–16.
[37] OMG, Meta Object Facility (MOF) Specification, Version 1.4, http://www.omg.org/technology/documents/formal/mof.htm, accessed in April 16, 2006.
[38] OMG, MOF 2.0 Versioning and Development Lifecycle RFP, http://www.omg.org/cgi-bin/doc?ad/02-06-23, accessed in April 16, 2006.
[39] OMG, Software Process Engineering Metamodel (SPEM), Version 1.1, http://www.omg.org/technology/documents/formal/spem.htm, accessed in April 16, 2006.
[40] OMG, Unified Modeling Language (UML) Specification, Version 1.5, Object Management Group, 2003.
[41] OMG, XML Metadata Interchange (XMI) Specification, Version 2.1, http://www.omg.org/technology/documents/formal/xmi.htm, accessed in April 16, 2006.
[42] R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 2005.
[43] H. Render, R. Campbell, An object-oriented model of software configuration management, in: International Workshop on Software Configuration Management, Trondheim, Norway, 1991, pp. 127–139.
[44] T. Roche, L.C. Whipple, Essential SourceSafe, Hentzenwerke Publishing, 2001.
[45] A. Sarma, Z. Noroozi, A.v.d. Hoek, Palantír: Raising awareness among configuration management workspaces, in: International Conference on Software Engineering, Portland, Oregon, 2003, pp. 444–454.
[46] J.S. Shirabad, T. Lethbridge, S. Matwin, Supporting software maintenance by mining software update records, in: International Conference on Software Maintenance, Florence, Italy, 2001, pp. 22–31.
[47] Sparx Systems, Enterprise Architect. http://www.sparxsystems.com/products/ea.html, accessed in April 16, 2006.
[48] J. Voelcker, Automating software: Proceed with caution, IEEE Spectrum 25 (7) (1988) 25–27.
[49] C.M.L. Werner, M.A.S. Mangan, L.G.P. Murta, R.P. Souza, M. Mattoso, R.M.M. Braga, M.R.S. Borges, OdysseyShare: An environment for collaborative component-based development, in: IEEE Conference on Information Reuse and Integration, Las Vegas, USA, 2003, pp. 61–68.
[50] B.A. White, Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction, Addison-Wesley, 2000.
[51] E.J. Whitehead, An Analysis of the Hypertext Versioning Domain, Ph.D. Thesis, University of California, Irvine, 2000.
[52] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering—An Introduction, Kluwer Academic Publishers, Norwell, Massachusetts, 2000.
[53] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, in: International Conference on Software Engineering, Edinburgh, Scotland, 2004, pp. 563–572.